



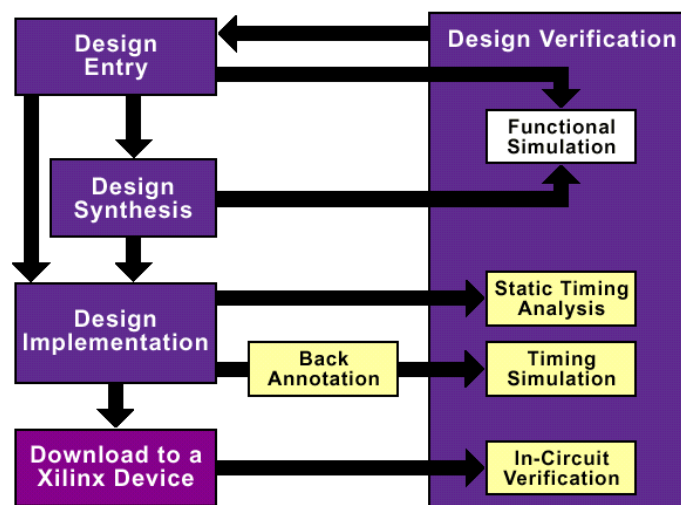
# Synteza logiczna

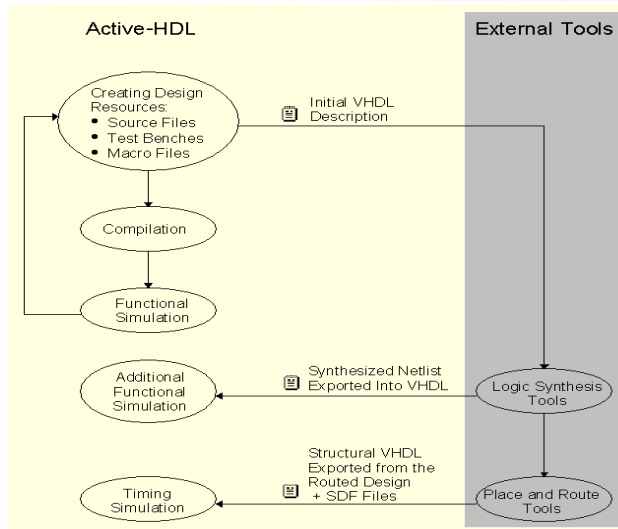


## Program wykładu

- **Wstęp do syntezy**
- **Sprzętowa reprezentacja obiektów w VHDL**
- **Synteza elementów podstawowych**
- **Synteza układów złożonych**
- **Synteza typów**
- **Konstrukcje niesyntezywalne**
- **Reguły i dobre praktyki**

- „*A VHDL Synthesis primer*” J.Bhasker,
- „*VHDL A Logic Synthesis Approach*” D.Naylor, S.Jones,
- „*VHDL Coding and Logic Synthesis with SYNOPSIS*”  
W.F.Lee,
- „*Reuse Methodology Manual*” M.Keating, P.Bricaud,
- „*Synthesis and Simulation Design Guide*” (*Xilinx manual*)
- „*Xilinx Synthesis Technology (XST) User Guide*”  
(*Xilinx manual*)
- „*Projektowanie układów cyfrowych z wykorzystaniem  
języka VHDL*” M.Zwoliński





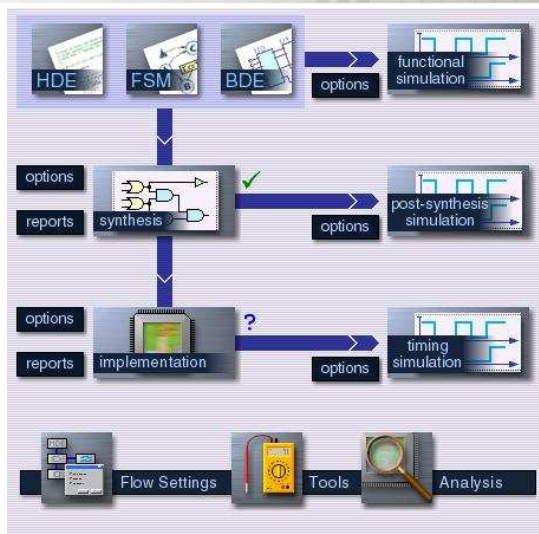
**Zewnętrzne narzędzia do syntezy i implementacji**

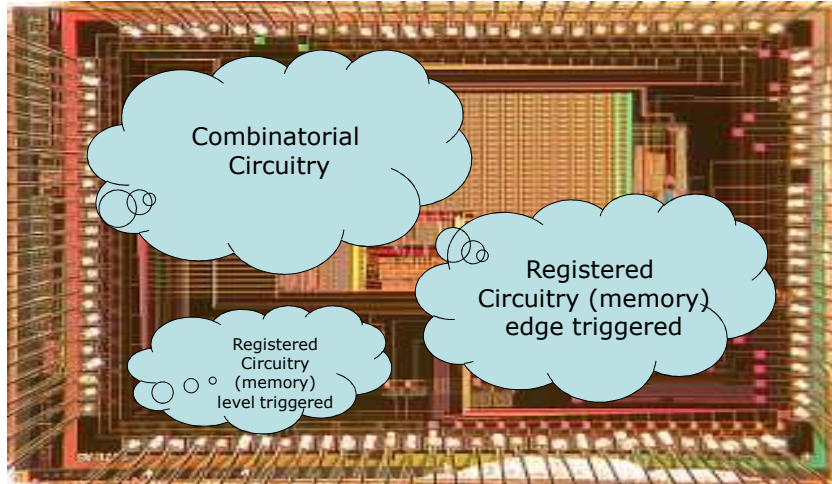
Graficzny *shell* po wybraniu opcji:

*Tools/Preferences/Environment/Flows/Integrated Tools*

oraz

*View/Flow*





**Obiekt - sygnał lub zmienna - może być reprezentowany jako:**

- **przerzutnik**  
(element pamięciowy wyzwalany zboczem)
- **zatrask**  
(element pamięciowy wyzwalany poziomem)
- **przewód**  
(element kombinacyjny)





## Synteza Sprzętowa reprezentacja obiektów

```
if C = 1 and
  C' event
then
  Q <= A;
end if;
```

**sygnał Q:**  
przerzutnik

```
if C = 1 then
  Q <= A;
else
  -- do nothing
end if;
```

**sygnał Q:**  
zatrzask

```
if C = 1 then
  Q <= A;
else
  Q <= B;
end if;
```

**sygnał Q:**  
multiplekser



## Synteza elementów podstawowych Operatory logiczne i arytmetyczne

```
architecture behavioral of AND_gate is
begin
  process (A, B)
  begin
    if A = '1' and B = '1' then
      X <= '1';
    else
      X <= '0';
    end if;
  end behavioral;
```

```
architecture inferred of AND_gate is
begin
  X <= A and B;
end inferred;
```

**Operatory** and or xor not nor nand xnor  
**implikowane:** + - \* / = /= > >= < <= ...

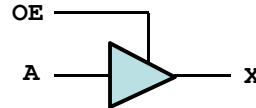


## Synteza elementów podstawowych Bufor trójstanowy

```

architecture behavioral of TRI_STATE_buffer is
begin
  process (A, OE)
  begin
    if OE = '1' then
      X <= A;
    else
      X <= 'Z';
    end if;
  end behavioral;

```



```

architecture inferred of TRI_STATE_buffer is
begin
  X <= A when OE = '1' else 'Z';
end inferred;

```



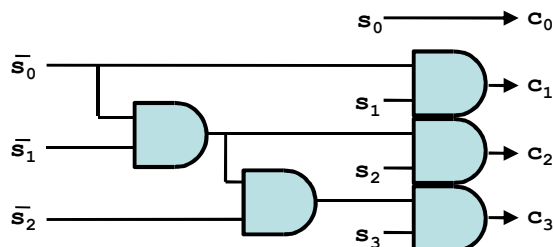
## Synteza elementów podstawowych Enkoder priorytetowy

```

architecture concurrent of PRIORITY_encoder is
begin
  code <= "0001" when sel(0) = '1' else
          "0010" when sel(1) = '1' else
          "0100" when sel(2) = '1' else
          "1000" when sel(3) = '1' else
          "0000";
end concurrent;

```

$$\begin{aligned}
 C_0 &= S_0 \\
 C_1 &= \bar{S}_0 S_1 \\
 C_2 &= \bar{S}_0 \bar{S}_1 S_2 \\
 C_3 &= \bar{S}_0 \bar{S}_1 \bar{S}_2 S_3
 \end{aligned}$$

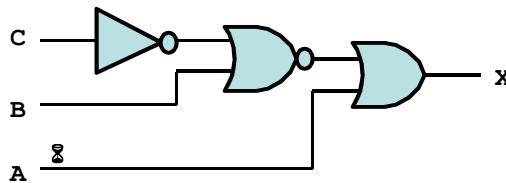


**Uwaga na funkcjory logiczne realizowane przy pomocy LUT !**



## Synteza elementów podstawowych Enkoder priorytetowy

```
architecture behavioral of PRIORITY_encoder is
begin
  process (A, B, C)
  begin
    if A = '1' then           -- A: sygnał przychodzący
      X <= '1';              -- z dużym opóźnieniem
    elsif B = '1' then
      X <= '0';
    elsif C = '1' then
      X <= '1';
    else
      X <= '0';
    end if;
  end behavioral;
end
```



**Uwaga na funkcjory logiczne realizowane przy pomocy LUT !**



## Sprzętowa reprezentacja obiektów Konstrukcje implikujące synteze przerzutnika D

### Proces zawierający listę wrażliwości:

```
process (CLK)
begin
  if CLK = '1' and CLK'event then           --rising edge
    Q <= D;
  end if;
end process;
```

### Proces zawierający klauzule wait:

```
process
begin
  wait until not CLK = '1' and CLK'event;   --falling edge
  Q <= D;
end process;
```

### Nie należy łączyć detekcji zbocza z innymi warunkami:

```
if CLK'event and CLK = '1' and CE = '1' then ...
```

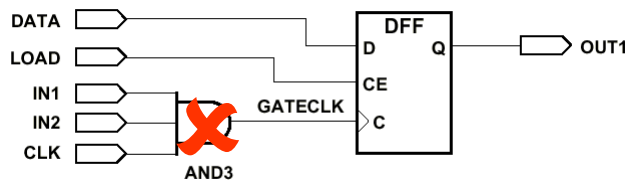
### Współbieżne przypisanie warunkowe (niezalecane):

```
Q <= D when CLK'event and CLK = '1';
```



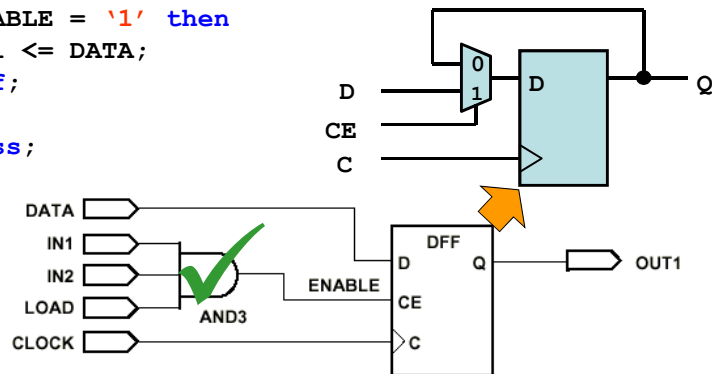
## Sprzętowa reprezentacja obiektów Przerzutnik z wejściem blokującym zegar

```
GATECLK <= IN1 and IN2 and CLK;  
process (GATECLK)  
begin  
  if GATECLK'event and GATECLK = '1' then  
    if LOAD = '1' then  
      OUT1 <= DATA;  
    end if;  
  end if;  
end process;
```



## Sprzętowa reprezentacja obiektów Przerzutnik z wejściem blokującym zegar

```
ENABLE <= IN1 and IN2 and LOAD;  
process (CLK)  
begin  
  if CLOCK'event and CLOCK = '1' then  
    if ENABLE = '1' then  
      OUT1 <= DATA;  
    end if;  
  end if;  
end process;
```







## Sprzętowa reprezentacja obiektów Konstrukcje implikujące Set / Reset

AGH

```
process (CLK)
begin
  if CLK and CLK'event then
    if SET = '1' then      -- SET synchroniczny
      Q <= '1';
    else
      Q <= D;
    end if;
  end if;
end process;

process (CLK, RESET)      -- RESET w sensitivity list!
begin
  if RESET = '1' then    -- RESET asynchroniczny
    Q <= '0';
  elsif CLK and CLK'event then
    Q <= D;
  end if;
end process;
```



## Sprzętowa reprezentacja obiektów Konstrukcje implikujące synteze zatrasku D

AGH

### Proces zawierający listę wrażliwości:

```
process (CLK, A, B)      -- list of all signals...
begin                  -- ...used in the process
  if CLK = '1' then    -- active high
    Q <= D;
  end if;
end process;
```

### Współbieżne przypisanie warunkowe (niezalecane):

```
Y <= D when CLK = '1';
```



## Sprzętowa reprezentacja obiektów Niezamierzona synteza zatrzasku

AGH

```
process (A, B, C, SEL)
begin
  if SEL = "00" then
    Y <= A;
  elsif SEL = "01" then
    Y <= B;
  elsif SEL = "10" then
    Y <= C;
  end if;
end process;
```

### Problem:

implikacja zatrzasku na skutek niepełnej specyfikacji wartości (dla SEL = "11" domyślnie przyjęta zostanie ostatnia wartość, co spowoduje konieczność zastosowania elementu pamięciowego).

```
process (A, B, C, SEL)
begin
  if SEL = "00" then
    Y <= A;
  elsif SEL = "01" then
    Y <= B;
  elsif SEL = "10" then
    Y <= C;
  elsif SEL = "11" then
    Y <= '0';
  end if;
end process;
```

### Rozwiązanie:

wyspecyfikowanie wszystkich możliwości (także w konstrukcji case – można tu użyć specyfikacji domyślnej przy pomocy klauzuli when others).

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

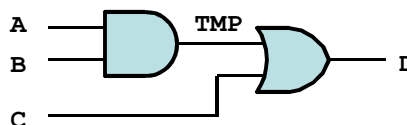
19



## Sprzętowa reprezentacja obiektów Sygnały a zmienne

AGH

```
signal A,B,C,D: bit
...
NO_MEMORY: process (A,B,C)
  variable TMP: bit;
begin
  TMP := A and B;
  D <= TMP or C;
end process;
```



Synteza TMP : przewód

```
signal A,B,C: bit
...
IS_IT_LATCH: process (A,B,C)
  variable TMP: bit;
begin
  C <= TMP and B;
  TMP := A or C;
end process;
```

Synteza TMP : zatrzask ?!

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

20



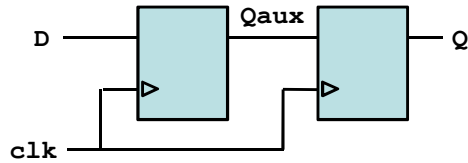
## Sprzętowa reprezentacja obiektów Sygnały a zmienne

AGH

```

signal Qaux: ...
FFx2: process (clk)
begin
...
Qaux <= D;
Q <= Qaux;
end process;

```

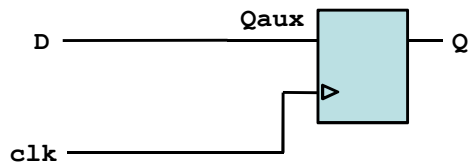


Synteza : 2 przerzutniki

```

FFx1: process (clk)
variable Qaux: ...
begin
...
Qaux := D;
Q <= Qaux;
end process;

```



Synteza : 1 przerzutnik

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

21



## Synteza układów złożonych Operatory logiczne

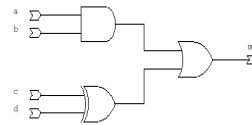
AGH

```

entity logical_ops is
port (a, b, c, d: in bit; m: out bit);
end logical_ops;

architecture example of logical_ops is
signal e: bit;
begin
m <= (a and b) or e;
e <= c xor d;
end example;

```

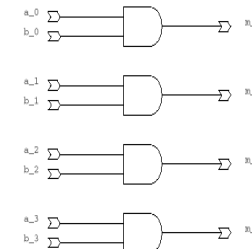


```

entity logical_bit is
port (a, b: in bit_vector (0 to 3);
m: out bit_vector (0 to 3));
end logical_bit

architecture example of logical_bit is
begin
m <= a and b;
end example;

```



Rajda & Kasperek © 2018 Katedra Elektroniki AGH

22

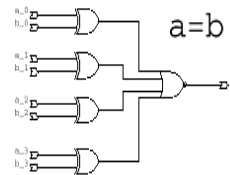


## Synteza układów złożonych Operatory relacji

AGH

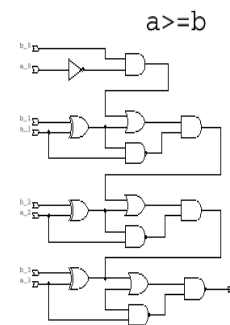
```
entity relational_equ is
  port (a, b: in bit_vector (0 to 3);
        m: out boolean);
end relational_equ;

architecture example of relational_equ is
begin
  m <= a = b;
end example;
```



```
entity relational_mag is
  port (a, b: in integer range 0 to 15;
        m: out boolean);
end relational_mag;

architecture example of relational_mag is
begin
  m <= a >= b;
end example;
```



Rajda & Kasperek © 2018 Katedra Elektroniki AGH

23



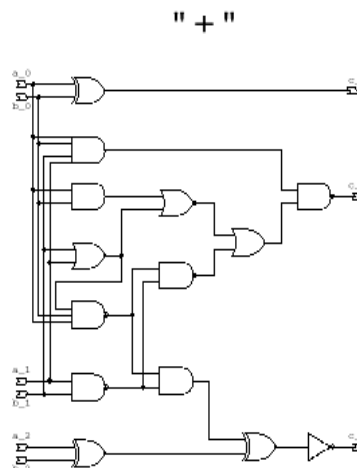
## Synteza układów złożonych Operatory arytmetyczne

AGH

```
package example_arithmetic is
  type small_int is range 0 to 7;
end example_arithmetic;
use work.example_arithmetic.all;

entity arith is
  port (a, b: in small_int;
        m: out small_int);
end arith;

architecture example of arith is
begin
  m <= a + b;
end example;
```



**Uwaga na *hard-makra* sumatorów (i mnożarek!!)**

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

24

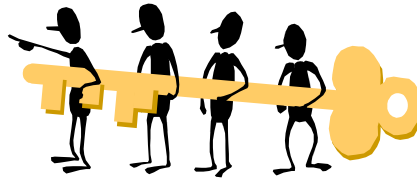


VHDL - wyrażenia sekwencyjne (*sequential*)

- warunkowe (*conditional signal assignment*): **if...**
- decyzyjne (*selected signal assignment*): **case...**

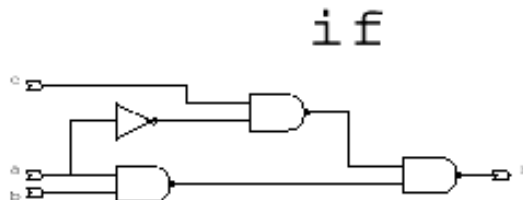
VHDL - wyrażenia współbieżne (*concurrent*)

- warunkowe (*conditional signal assignment*): **when...**
- decyzyjne (*selected signal assignment*): **with...**



```
entity control_stmts is
port (a, b, c: in boolean;
      m: out boolean);
end control_stmts;
```

```
architecture example of control_stmts is
begin
  process (a, b, c)
  variable n: boolean;
  begin
    if a then
      n := b;
    else
      n := c;
    end if;
    m <= n;
  end process;
end example;
```





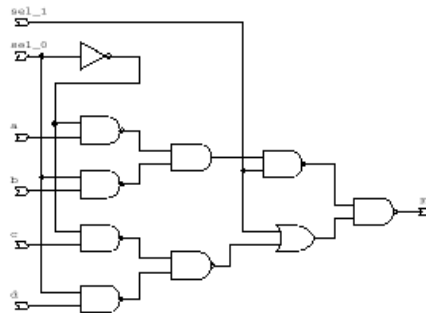
## Synteza układów złożonych Sekwencyjne instrukcje wyboru: **case**

AGH

```
entity control_stmts is
port (sel: in bit_vector (0 to 1); a,b,c,d: in bit;
      m: out bit);
end control_stmts;
```

```
architecture example of control_stmts is case
begin
```

```
  process (sel,a,b,c,d)
  begin
    case sel is
      when b"00" => m <= c;
      when b"01" => m <= d;
      when b"10" => m <= a;
      when others => m <= b;
    end case;
  end process;
end example;
```



Rajda & Kasperek © 2018 Katedra Elektroniki AGH

27



## Synteza układów złożonych Współbieżne instrukcje wyboru: **when / with**

AGH

```
entity control_stmts is
port (a, b, c: in boolean; m: out boolean);
end control_stmts;
architecture example of control_stmts is
begin
  m <= b when a else c;
end example;
```

```
entity control_stmts is
port (sel: in bit_vector(0 to 1); a,b,c,d: in bit; m: out bit);
end control_stmts;
architecture example of control_stmts is
begin
  with sel select
    m <= c when b"00",
    m <= d when b"01",
    m <= a when b"10",
    m <= b when others;
end example;
```

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

28



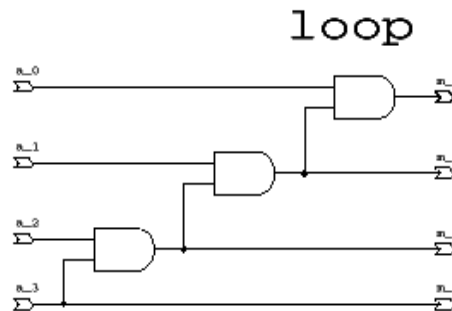
Synteza układów złożonych  
Konstrukcje "powielające" logikę

AGH

```
entity loop_stmt is
port (a: bit_vector (0 to 3);
      m: out bit_vector (0 to 3));
end loop_stmt;
```

Nie ma konieczności  
deklarowania indeksu i !

```
architecture example of loop_stmt is
begin
process (a)
variable b: bit;
begin
b := '1';
for i in 0 to 3 loop
b := a(3-i) and b;
m(i) <= b;
end loop;
end process;
end example;
```



Synteza układów złożonych  
Konstrukcje "powielające" logikę

AGH

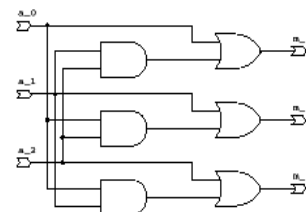
```
entity subprograms is
port (a: bit_vector (0 to 2);
      m: out bit_vector (0 to 2));
end subprograms;
```

```
architecture example of subprograms is
```

```
function simple (w, x, y: bit) return bit is
begin
return (w and x) or y;
end;
```

```
begin
process (a)
begin
m(0) <= simple(a(0), a(1), a(2));
m(1) <= simple(a(2), a(0), a(1));
m(2) <= simple(a(1), a(2), a(0));
end process;
end example;
```

subprogram



**Sekwencyjny (*shift register*):**

- z operatorem połączenia (&)  
shreg <= shreg (6 downto 0) & SI;
- z pętlą for...loop  
for i in 0 to 6 loop  
shreg(i+1) <= shreg(i);  
end loop;  
shreg(0) <= SI;
- z operatorami przesuwu (sll, srl, ...)

**Kombinacyjny (*barrel shifter*):**

- z operatorami przesuwu (sll, srl, ...)  
with SEL select  
SO <= DI when "00",  
DI sll 1 when "01",  
DI sll 2 when "10",  
DI sll 3 when others;
- z operatorem połączenia (&)

- implikowane (*inferred*) lub podstawiane (*instantiated*)
- implementowane jako rozproszone lub blokowe (w zal. od rozmiaru, szybkości i zajętego obszaru)
- synchroniczne (ew. z asynchronicznym odczytem – rozproszone)
- RAM (także inicjalizowane) lub ROM
- wykorzystywane także dla logiki kombinacyjnej i FSM

Method	Advantages	Disadvantages
Inference	<ul style="list-style-type: none"> <li>• Most generic way to incorporate RAMs into the design, allowing easy/automatic design migration from one FPGA family to another</li> <li>• FAST simulation</li> </ul>	<ul style="list-style-type: none"> <li>• Requires specific coding styles</li> <li>• Not all RAMs modes are supported</li> <li>• Gives you the least control over implementation</li> </ul>
CORE Generator software	Gives more control over the RAM creation	<ul style="list-style-type: none"> <li>• May complicate design migration from one FPGA family to another</li> <li>• Slower simulation comparing to Inference</li> </ul>
Instantiation	Offers the most control over the implementation	<ul style="list-style-type: none"> <li>• Limit and complicates design migration from one FPGA family to another</li> <li>• Requires multiple instantiations to properly create the right RAM configuration</li> </ul>





## Synteza typów Typ `integer`

- Typy i podtypy, które zawierają wartości ujemne w swojej definicji zakresu, kodowane są w kodzie uzupełnień do 2.
- Typy i podtypy, które zawierają tylko wartości dodatnie, kodowane są w naturalnym kodzie binarnym.
- Liczba użytych bitów zależy od największej dopuszczalnej wartości dla danego obiektu.

```
-- binary encoding having 7 bits
type int0 is range 0 to 100;
type int1 is range 10 to 100;

-- 2's complement encoding having 8 bits (including sign)
type int2 is range -1 to 100;

-- binary encoding having 3 bits
subtype int3 is int2 range 0 to 7;
```



## Synteza typów Typ `integer`

```
type short is integer 0 to 255;
subtype shorter is short range 0 to 31;
subtype shortest is short range 0 to 15;

signal op1, op2, res1: shortest;
signal res2: shorter;
signal res3: short;
begin
  res1 <= op1 + op2; -- truncate carry
  res2 <= op1 + op2; -- use carry
  res3 <= op1 + op2; -- use carry and zero extend
```



## Synteza typów Typy wyliczeniowe

### Deklaracja:

### Jest syntezowana jako:

```
type direction is (left, right, up, down);      -- two wires
type cpu_op is (execute, load, store);          -- two wires
                                                -- the encoding of 11 is a "don't care"

subtype mem_op is cpu_op range load to store;   -- two wires
                                                -- the encodings of 00 and 11 are "don't cares"

-- User Defined Encoding
attribute enum_encoding: string;
attribute enum_encoding of cpu_op: type is
  "001" &    -- execute
  "010" &    -- load
  "100";     -- store
```



## Synteza typów Typy wyliczeniowe i pozostałe

- Typy wyliczeniowe w procesie syntezy domyślnie kodowane są binarne. Do poszczególnych elementów (L) typu asygnowane są kolejne wartości, przy czym pierwszy od lewej otrzymuje wartość zero.
- Liczba elementów (N) obiektu reprezentującego typ wyliczeniowy będzie najmniejszą z możliwych liczbą, spełniającą warunek:  $L \leq 2^N$
- Typy **bit** i **boolean** są syntezowane jako „wire”
- Typ **character** jest syntezowany jako  $8 \times$  „wire”



## Synteza typów Typ `std_logic_1164`

AGH

### Zalecany, bo:

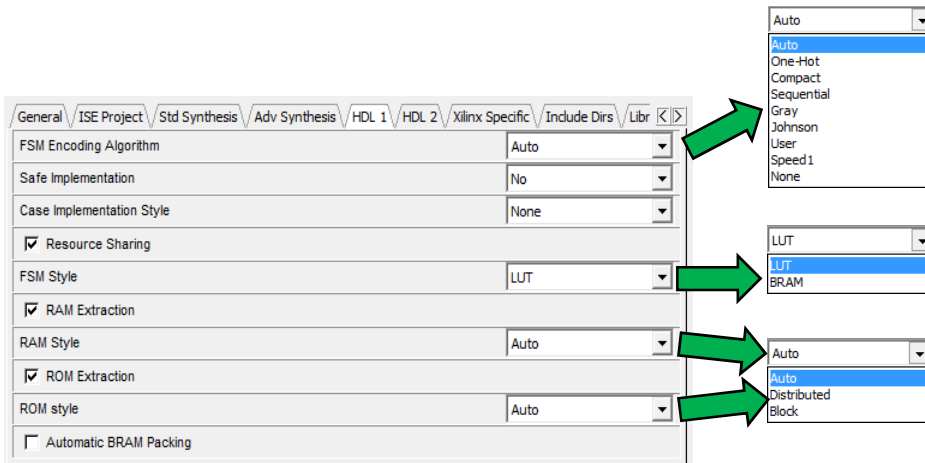
- duża liczba wartości (9), reprezentujących większość rzeczywistych stanów w układach cyfrowych,
- automatyczna inicjalizacja do wartości 'U' – zmusza projektanta do zainicjowania projektu *ex plicite*. Nie należy omijać tej cechy przez inicjalizację sygnałów i zmiennych w momencie ich deklaracji – rezultatem takiego postępowania może być otrzymanie w wyniku syntezy układu, który nie daje się zainicjować!
- łatwa integracja z innymi modułami – np. typ `integer` może zostać zsyntezowany, ale przy symulacji będzie wymagał wykonywania czasochłonnych funkcji konwersji.
- **po syntezie i implementacji tylko typ `std_logic`**



## Synteza automatów Algorytmy kodowania stanów

AGH

- *Auto*  
Najlepszy, dostosowany do automatu algorytm kodowania.
- *One-Hot*  
Zawsze włączony tylko jeden przerzutnik. Odpowiedni dla FPGA (duża liczba przerzutników) oraz optymalizacji prędkości i poboru mocy.
- *Compact*  
Minimalizuje liczbę przerzutników. Optymalny dla minimalizacji obszaru.
- *Sequential*  
Identyfikuje długie ścieżki i nadaje w nich kolejne wartości binarne.
- *Gray*  
Zawsze tylko jedna zmienna zmienia swoją wartość. Właściwy dla automatów o długich ścieżkach bez odgałęzień. Minimalizuje hazardy i szpilki.
- *Johnson*  
Podobnie jak Gray.
- *User*  
Sposób definiowany przez użytkownika przy pomocy typu wyliczeniowego.
- *Speed1*  
Optymalizowany dla szybkości działania. Liczba rejestrów zależna od konkretnego automatu, ale zwykle większa niż liczba stanów.



- **klauzule czasowe:**
  - przypisać (**after**, **transport**, **inertial**)
  - **wait for**
- **zmiennoprzecinkowe typy danych (**real**)**
- **operacje na plikach – ograniczone:**
  - **read**: inicjalizacja pamięci z pliku
  - **write**: debugowanie



Synteza

## Kodowanie uwzględniające proces syntezy

AGH

1. Rejestry wyjściowe dla sygnałów.
2. Jeden atrybut sygnał **'event'** na jeden proces.
3. Sterowanie sygnału tylko z jednego procesu (jeżeli występuje potrzeba kontroli z 2 lub więcej procesów – ok – ale wtedy potrzebny dodatkowa instrukcja arbitrażu).
4. Odpowiedni podział modułów logiki kombinacyjnej.
5. Podział na bloki z uwagi na wymaganą szybkość działania (*speed versus area optimisation*).
6. Rozdzielenie bloków logiki asynchronicznej i synchronicznej (instrukcja **process** a instrukcje współbieżne).
7. Odpowiednie rozwiązania dla modułów realizujących funkcje arytmetyczne.
8. Grupowanie sygnałów pod kątem czasu syntezy.
9. Unikanie „*multicycle paths*” – czyli ścieżek sygnałów, które odchodzą od reguły „jednego okresu”.
10. Nie testowanie sygnałów na wartość 'X' 'Z'.
11. Asynchroniczny reset – jeżeli ten sygnał dostępny z zewnątrz. Nie generujemy własnych asynchronicznych sygnałów/resetów.

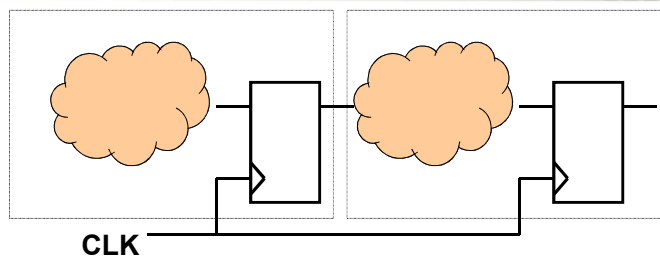
Rajda & Kasperek © 2018 Katedra Elektroniki AGH

41



## Kodowanie uwzględniające proces syntezy

AGH



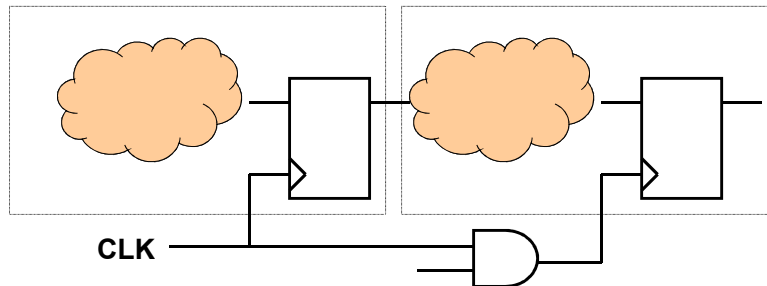
Modelowa struktura projektu:

- Jeden sygnał zegarowy
- Wszystkie przerzutniki wyzwalane tym samym zboczem

Problemy przy dwóch aktywnych zboczach:

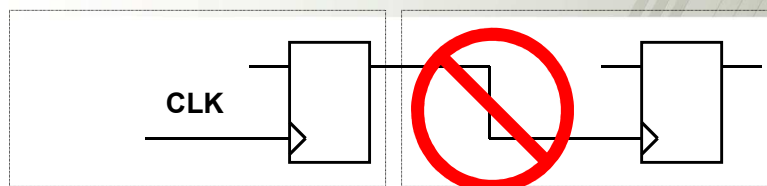
- Zależność od współczynnika wypełnienia (tolerancja na zmiany *duty cycle* w dokumentacji projektu!)
- Problemy z metodą testowania typu ścieżka brzegowa (JTAG 1149)

Rajda & Kasperek © 2018 Katedra Elektroniki AGH

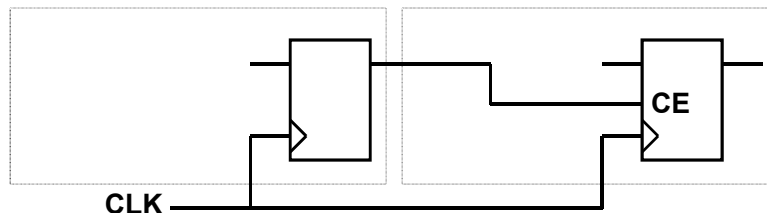


Asynchroniczne kluczowanie zegara - **same problemy!**  
(niewykorzystanie zasobów dystrybucji sygnałów zegarowych, problemy z testowaniem, gorsze parametry czasowe itp.)

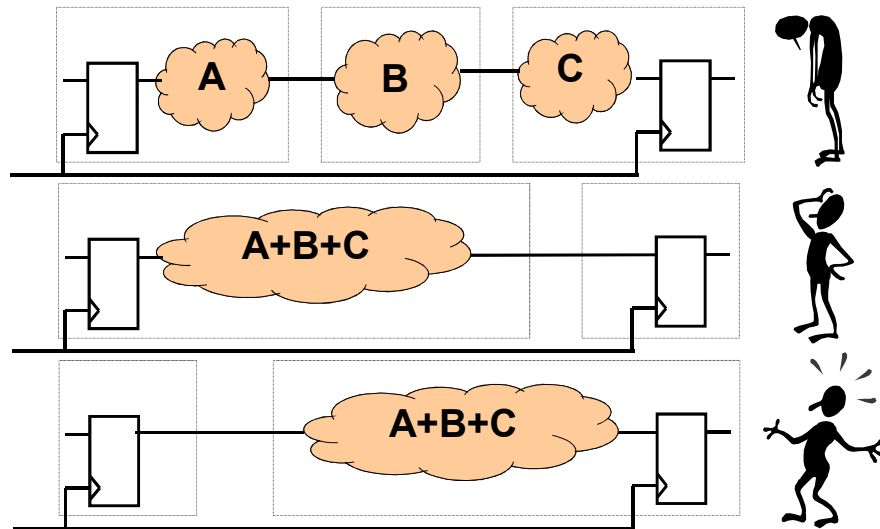
Rozwiązanie – stosowanie wejść CE – kodowanie przez warunek:  
*if ce='1' po sekwencji if clk'event and clk='1'.*



Nie należy stosować wewnętrznie (asynchronicznie!) generowanych sygnałów zegarowych.



Należy natomiast projektować układy synchroniczne lub używać kilku sygnałów zegarowych (patrz: DLL).



Language Assistant

VHDL

- Synthesis templates
  - Adders
    - Accumulator
    - Adder with carry in
    - Adder with carry out
    - Simple adder/subtractor
  - Comparators
    - Equality comparator
    - 'Greater-than' comparator
    - 'Less-than' comparator
    - 'Not-greater-than' comparator
    - 'Not-less-than' comparator
    - Unequality comparator
  - Counters
    - Asynchronous binary counter
    - Prescaler
    - Synchronous binary counter (for INTEG)**
    - Synchronous binary counter (for STD\_LC)
    - Synchronous gray counter (for STD\_LOC)
  - Decoders

```

--Synchronous binary up/down counter (for INTEGERS)
--CLK: in STD LOGIC;
--RESET: in STD LOGIC;
--CE, LOAD, DIR: in STD LOGIC;
--DIN: in INTEGER range 0 to 2**WIDTH-1;
--COUNT: out INTEGER range 0 to 2**WIDTH-1;

-- The declaration of the auxiliary signal 'COUNTER' must be inserted in
-- the architecture declarative part:
-- signal COUNTER: INTEGER range 0 to 2**WIDTH-1;

process (CLK, RESET)
begin
    if RESET='1' then
        COUNTER <= 0;
    elsif CLK='1' and CLK'event then
        if LOAD='1' then
            COUNTER <= DIN;
        else
            if CE='1' then
                if DIR='1' then
                    if COUNTER = 15 then
                        COUNTER <= 0;
                    else

```

Ciąg dalszy  
nastąpi...

